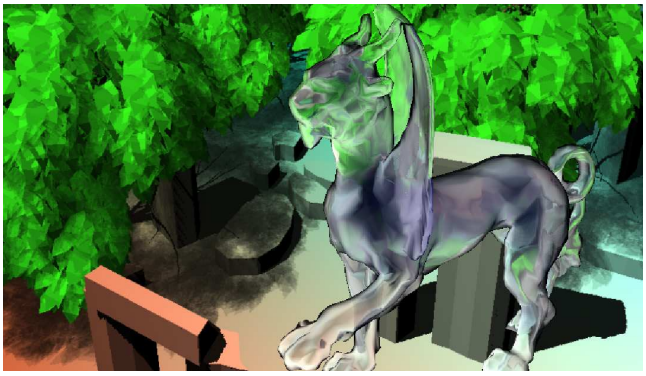
Fast Scene Voxelization and Applications

Elmar Eisemann, Xavier Décoret

Fast Scene Voxelization and Applications

Elmar Eisemann*
ARTIS-GRAVIR/IMAG-INRIA†/ ENS‡

Xavier Décoret§
ARTIS-GRAVIR/IMAG-INRIA



Abstract

This paper presents a novel approach that uses graphics hardware to dynamically calculate a voxel-based representation of a scene. The voxelization is obtained on run-time in the order of milliseconds, even for complex and dynamic scenes containing more than 1,000,000 polygons. The voxelization is created and stored on the GPU avoiding unnecessary data transfer. The approach can handle both regular grids and locally optimized grids that better fit the scene geometry. The paper demonstrates applications to shadow calculation, refraction simultation and shadow volume culling/clamping.

Keywords: voxelization, shadows, GPU, refraction

1 Introduction

Voxels have a long history in volume graphics and are still of importance in medical visualization to represent the acquired data from CT scans. Voxels are three dimensional entities that encode volumetric information, as opposed to boundary representations such as meshes which only describe the surface of objects. Voxel-based representation thus gives information about where matter is in space. Converting between voxels and boundary representations is a well studied problem. For example, marching cubes can extract a surface from a potential function defined over a voxel grid. In this paper, we present an efficient hardware based approach to the opposite task : constructing a voxel grid from an input mesh.

*e-mail: Elmar.Eisemann@inrialpes.fr
†ARTIS is a team of the GRAVIR/IMAG laboratory, a joint effort of CNRS, INRIA, INPG and UJF
‡École Normale Supérieure Paris
§e-mail:Xavier.Decoret@inrialpes.fr

The presentation is structured as follows. In Section 2 we present in details our texture based voxel representation along with an algorithm to compute it from a 3D scene. We discuss strengths and limitations of our method and compare it to related work. Section 3 presents three applications where this novel representation proves very beneficial. Finally we discuss the advantage of our approach given the evolution of graphic hardware and possible avenues of future research.

2 Principle of the slicemap

To voxelize a scene, a grid of cells is defined around it. The primitives are traversed and, for each of them, the cells they intersect are found. Our approach accomplishes this task efficiently with the graphics hardware, based on two observations. First, a rendered view of a scene implicitly defines a grid. The outline of that grid is given by the view frustum of the camera and its resolution is given by the resolution of the viewport and by the finite precision of the frame buffer. Second, when the graphic card renders a view, it does traverse every primitive and it does find the cells intersected in this implicit grid. Indeed, for every fragment produced during the rasterization of a primitive, the (x,y) pixel coordinates and z value indicate a cell. In classical rendering, the z value is used for hidden face removal and only the color of the closest fragment is kept. Although other fragments are discarded, the system has had access to it at some point. Our idea is to keep this information instead of discarding it and to encode it in the RGBA channels.

2.1 Grid encoding

We define a grid by placing a camera in the scene and adjusting its view frustum to enclose the area to be voxelized. The camera can be orthographic or perspective and can be placed at any position outside the zone of interest. Then, we associate a viewport to the camera. The (w,h) dimensions of that viewport indicate the resolution of the grid in the x and y directions. A pixel (x,y) represents a column in the grid. Each cell within this column is encoded via the RGBA value of the pixel. Instead of considering this value as 4 bytes typically encoded on 8 bits, we consider it as a vector of 32 bits, each one representing a cell in the column. The division of a column into 32 cells can be done in different ways. The simplest, most natural one is to evenly divide the range between the near and far planes but we will see how to improve on this in some situations. Once a camera, a viewport and a division scheme are defined, the corresponding image represents a $w \times h \times 32$ grid with one bit of information per cell. We will use that bit to indicate whether a primitive passes through a cell or not. Figure 1 shows an example of such a grid. The union for all columns of voxels corresponding to a given bit defines a *slice*. Consequently, the image/texture encoding the grid is called a *slicemap*.

2.2 Rasterization in the grid

To construct the slicemap from a polygonal scene using graphics hardware. We render it into a texture using a simple fragment

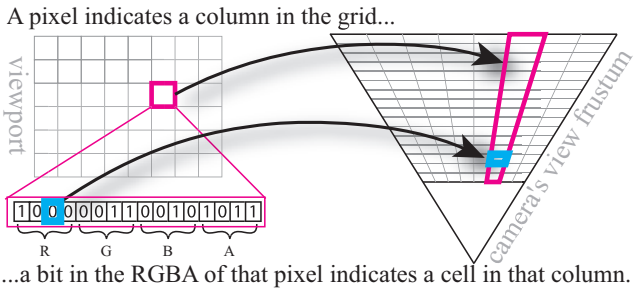


Figure 1: Encoding of a grid in the viewport of a camera. For clarity 4 bits per channel in the color buffer (16 slices) are assumed

shader. The projection, modelview and viewport matrices are set to match the chosen grid. The texture is initially cleared with black so all bits are set to 0.

We must find, for each primitive, the voxels that it intersects and set the corresponding bits to 1. Rasterizing the primitive will produce a single fragment for each of the columns intersected. The depth d of that fragment indicates in which slice it falls. We use a fragment program to transform this depth into a 32 bit mask with 0 everywhere except for a 1 in the bit corresponding to the slice. The depth values for fragments are in the range $[0,1]$ but the distribution is not uniform in world coordinates. Using this depth for slices would put too much resolution close to the near plane and not enough close to the far plane. Instead, we use the real distance to the camera's COP by transforming the 3D position of the vertex by the transformation matrices. This distance is passed to the fragment shader as texture coordinates. Due to on-surface interpolation, a fragment obtains a correct z in $[-z_n, z_f]$ which is then mapped linearly to $[0,1]$ using :

$$z' = \frac{z + z_n}{z_n + z_f} \tag{1}$$

This normalized distance is used to perform a texture lookup in a 1D texture that gives the 32 bits mask corresponding to the slice in which z' falls. Currently, the texture lookup is much more efficient than performing the arithmetic in the fragment program. The resulting texture will be referred to as the *cellmask texture*. Its format is RGBA with 8 bits per channel to represent the 32 slices. Note that it is independent of the actual voxel grid's position and is computed only once. It could even be included on the chip and provided as a function in shaders. Our convention for the cellmask texture implies that the values in the mask are between 2^0 for the nearest one and 2^{31} for the farthest cell.

The color/bitmask obtained from the texture must then be OR-ed with the color in the frame buffer to get the correct bitmask in the end. OpenGL's logical operation provides that functionality.

2.3 Uniform *vs.* local slicemap

As we mentioned earlier, there are various ways to divide a column of our grid in 32 cells. We just described a column-indepedent scheme that produces a *uniform* slicing, potentially wasting some resolution.

If the depth of fragments in a given column do not range from 0 to 1, we end up with useless cells in empty areas and cells too coarse to capture the details in other areas, as can be seen on Figure 2-a. Equation (1) reveals that z_n and z_f could be chosen for each column independently in order to enclose the geometry in this column more tightly. To perform this local fitting we recover the scene extent for each pixel seperately by rendering two depth maps.These renderings could also be done using a simple bounding geometry.

These two textures will be called *near* and *far depth textures*. We then generate the slicemap as before by rendering the scene, applying a modified shader so that eq.(1) now uses *local* values of z_n and z_f. Figure 2-b shows that it generally creates a finer voxelization, but locally voxels might be less fitting (compare voxel A in figure 2-b).

2.4 Grid's resolution

The number of columns in the grid is the resolution of the slicemap. Therefore it is limited by the maximum viewport size supported by the graphic card, currently 4096×4096. This (x,y) resolution is huge compared to the typical size of volumetric datasets, which rarely exceeds 512^3. On the other hand, the number of slices is 32, which is rather small. The limitation to 32 slices arises from the fact that logical operations can only be applied to 8 bit channels. Nevertheless there are different ways to bypass this limitation and increase the number of slices. The most natural one is to use additional slicemaps. Every extra map requires a supplementary rendering of the geometry so there cannot be too many of them or the performance would suffer. Fortunately, modern graphic cards allow a shader to draw to several color buffers at once by defining Multiple Render Targets (MRT). Even if this is formally equivalent to performing several renderings of the geometry with the same matrices, this is actually much more efficient as the transformation, assembly and rasterization stages are performed only once. Only a small amount of extra processing needs to be done in the fragment shader to decide on the value of the output colors.

In the future, if 32 bit logical operations become available, it will even be possible to go up to 4 MRT $\times$4 channels $\times$ 32 bits = 512 slices. Integer operations would also be welcome, but the pack function of the current shader model (four unsigned bytes can be merged bitwise in a 32 bit float) would already allow an easy encoding and decoding inside the shader.

2.5 Strength and Limitations

The slicemap has several benefits. First, it fits in one (or a few more, if MRT is used) texture. Thus it can be queried from vertex or fragment shaders and several useful applications are demonstrated in section 3. Second, it is generated by the hardware. Not only does this mean high performance, but also that the slicemap lives on the GPU and never needs to be transferred from or to the CPU. Moreover, it can be generated from arbitrary geometry that itself lives on the GPU (e.g. vertex buffer objects) and in particular geometry that is moved within a vertex shader (e.g. through skeleton animation). This means reduced bandwith, no copying of the geometry on the GPU and no CPU work to transform or animate the geometry. In

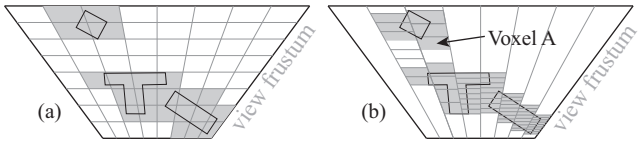


Figure 2: Uniform vs. local slicing (a) columns are sliced uniformly using the camera's depth range, yielding a "regular grid" and coarse voxels (b) each column is sliced using local depth range, yielding a "distorted grid" and generally finer voxelization.

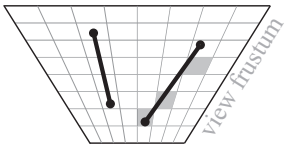


Figure 3: Limitations of the slicemap: the left primitive is completely missed as it is aligned with the view direction; the right one has a large slope in z and is not voxelized continuously.

practice, on a GeForce 6800 TD, we are able to voxelize a scene of 1,124,571 triangles in a $512 \times 512 \times 96$ (3 MRT) grid with uniform resolution at 70Hz. If we use local resolution, we need two extra rendering and the frame rate is 60Hz. As a comparison, the octree based approach of [Haumont and Warzee 2002] et al. takes ~ 3 min to intersect $\sim$ 100K polygons with an octree of maximum depth 8 i.e. a 256^3 grid. Of course the octree representation is much richer.

The efficiency of the slicemap encoding and generation has a price. We can only store one bit of information per voxel, typically the presence of matter, but we cannot store the color or number of objects in the voxel. It is possible to decrease the number of slices and allow a few bits per voxel as we will see in section 3.1 but this is a very limited tradeoff.

The other limitation of slicemap is that it only encodes the boundary representation (BRep) of the scene. We do not perform any floodfill to classify interior and exterior voxels. We only consider a subset of the voxels that are intersected by a primitive. Indeed, using rasterization to find the columns intersected by a primitive gives only a single fragment per column. Consequently, intersected voxels are not always found as can be seen on the right part of Figure 3.

The voxels found do not create a "continuous" set : there might be holes. They will occur when the slope of the primitive in the z-direction is too high. This is a well known problem when rasterizing in a grid [Bresenham 1965]. An even more extreme case can be encountered. If a primitive is perfectly aligned with the view direction, as shown on left of Figure 3, OpenGL will not produce any fragments and the primitive will be missed (a possible workaround for this situation is described in [Aila and Akenine-Möller 2005]). Depending on the application, this may or may not be an issue. An important point is that because the slicemap can have a large resolution in x and y directions, the impact of those holes is usually not dramatic. As more fragments are rasterized, the continuity in depth increases and due to the smaller number of slices, holes become rare.

To summarize, the slicemap allows for a fast, hardware assisted, approximate determination of boundary voxels for arbitrary scenes, together with a compact, GPU friendly encoding into textures.

2.6 Related work

The slicemap is an approximation of the different layers of matter visible from a camera. In a sense, it is related to Layered Depth Images (LDI) as described in [Shade et al. 1998]. LDIs are more general and store arbitrary information per layer but are much more costly to obtain, usually using image warping or ray-tracing in a pre-process. Hardware acceleration can be used by performing *depth peeling* as described in [Everitt 2001]. However, this requires several renderings of the scene. Potentially the number of layers depend on the viewpoint and should be fixed in advance to avoid varying framerates, when using occlusion queries. Detecting matter in a depth layer can also be achieved using occlusion queries [Lloyd et al. 2004] but once again this requires several renderings. Recently octrees have been constructed on the GPU [Lefebvre et al. 2005]. But instead of filling the octree with scene geometry, it is used as a structure to perform texture mapping and the stored information is rather sparse compared with the huge amount of geometry we process.

3 Applications

3.1 Transmittance Shadow Maps

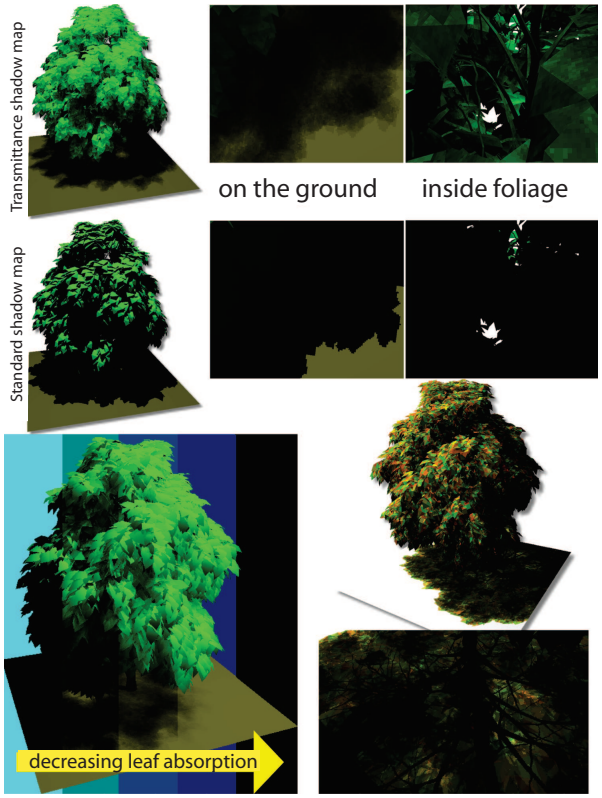


Figure 4: Upper part: difference between standard shadow map and our approach. Close ups on the ground emphasize the difference. Lower part left: A combined view of transmittance-based shadows with different absorption coefficient. Right: using 3 different leaf colors, transmittance shadow maps can achieve colored shadows. Notice the multi-colored shadows cast on the ground.(cf. color plate AII)

A standard shadow map [Williams 1978] stores the depth of the closest occluder along sample rays. Points along that ray are classified as shadowed or lit by comparing their depth against the stored one. Instead, a deep shadow map [Lokovic and Veach 2000] stores for each ray a one dimensional function describing the light intensity along the ray. This technique achieves realistic self shadowing for very complex volumetric structures like hair. Points are shaded continuously based on their position, by evaluating these functions.

Deep shadow maps account for three phenomena : transmittance of semi-transparent surfaces (e.g. tinted glass), partial occlusion of the light beam by thin occluders (e.g. hair strands) and volumetric extinction (e.g. fog). In this section, we described how slicemaps can be used to render the first effect. Partial occlusion could be handled, as done in general, using a higher resolution and interpolation.

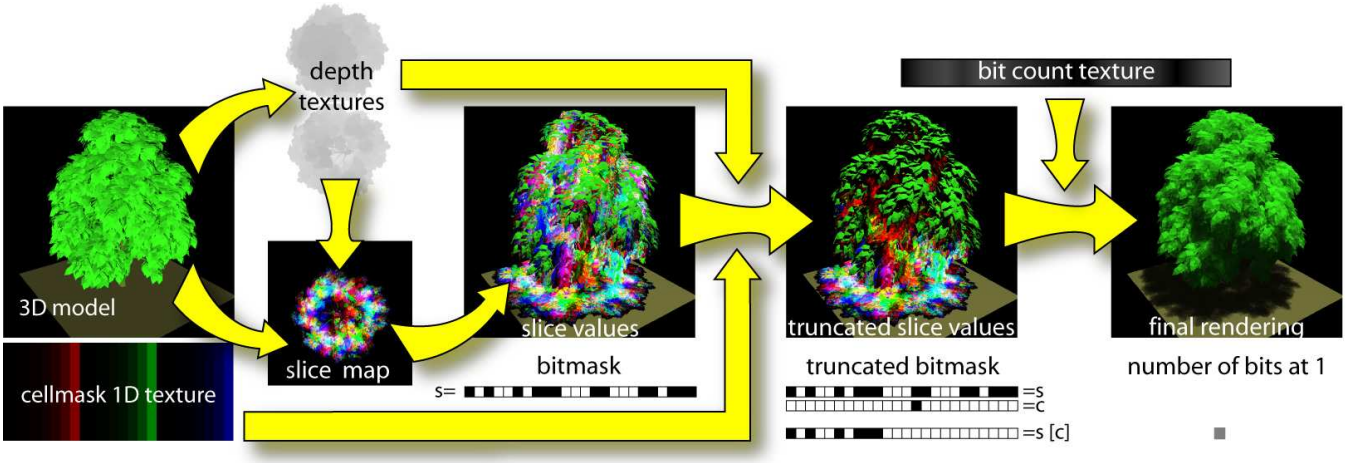


Figure 5: Overview of the transmittance shadow map algorithm - here with 24 slices stored in RGB channels of 8 bits (cf. color plate AI)

Section 3.3 will describe how they can be used to render volumetric effects. The approach in this section also relates to Opacity Maps [Kim and Neumann 2001], which do not aim at real time applications. The scene is decomposed into layers using several render passes to obtain a local opacity value, the values between the maps are interpolated linearly (which is also possible with our approach).

Let's consider the foliage of a tree lit by a point light source L. If we neglect indirect illumination and refraction, the irradiance at a point P on a leaf is found by tracing the ray to the light source and summing the contributions of all traversed leaves. If we assume all leaves attenuate the light with the same factor, it amounts to counting the number of leaves intersecting the segment $[PL]$. For that, we render a local slicemap from the light source. To shade a point P at depth z, we do a projective texture lookup into the depth textures to get z_n and z_f. If $z \leq z_n$ the point is fully lit. If $z_n < z$, we first retrieve the cellmask $c = 2^i$ such that P lies in cell i (cf. section 2.2). Then we do a projective texture lookup to get the bitmask s from the slicemap. In that bitmask, we must set all bits j to 0 with $j > i$. Indeed, such bits correspond to cells further from L than P (cf. section 2.2). Mathematically, this corresponds to a modulo operation $\bar{s} = s \mod c$ (e.g. `fmod` in a Cg shader). Finally we need to compute the number of 1s in $\bar{s}$. Once again we do this using texture lookups in a texture we will refer to as *summask texture*.

The shading is computed as $(1-\sigma)^n$ where σ is the attenuation factor for a leaf. The computation of $(1-\sigma)^n$ can actually be baked into the summask texture (values must then be multiplied instead of added) to save some instructions in the shader. However, passing σ as a uniform parameter allows to dynamically change its value. The approach is summarized on Figure 5.

We implemented this algorithm on a GeForce 6800FX Ultra. We used a resolution of 512×512 and 3 MRT buffers which gives 96 slices. Our system combines semi-transparent objects and opaque objects (e.g. the trunk of the tree). For that, the shader does an extra lookup in a standard shadowmap generated with only opaque objects. If the point is not shadowed, we continue with the transmittance shadow map generated with only the semi-transparent parts. We tested it on a tree model containing 160,653 polygons. Figure 4 shows the drastic difference between our transmittance shadow mapping and standard shadow mapping. Note the variation of shadow intensity in the foliage which makes the shape of the tree a lot more perceptible. Attenuation effects can also be observed on the ground and can be changed dynamically by varying σ. There are fewer leaves close to the silhouette of the tree thus the shadow becomes less pronounced. The rightmost images show

chestnut tree	SSM	TSM	
		uniform	local
frame rate	128	60/50/40	37/29/24
opaque map	3ms	<1/<1/<1 ms	<1/<1/<1 ms
near map	-	3/3/3 ms	3/3/3 ms
far map	-	-	2/2/2 ms
slicemap	-	2/4/7 ms	7/9/14 ms
shading	5	14/16/18 ms	20/28/32 ms

Table 1: Frame rate (Hz) and timings (ms) for standard shadow mapping and transmittance shadow mapping (TSM) with uniform and local slicing. For TSM, we give the timings for 1,2 and 3 MRT. Near,far and slicemaps are computed only for the transparent geometry. (tree model contains 160,653 polys (1,493 opaque))

an interesting variation where we tradeoff slicemap resolution for increased number of bits (therefore information) per voxel. Simply put, using 3 MRTs we can make a 32 slices slicemap for green, reddish and yellowish leafs in a single pass.

The system turns in real-time as shown on table 1. It scales well with the geometry, as it is mostly pixel shader bounded. In particular, we evaluate the shader for hidden fragments. For a model with high depth complexity such as our tree, deferred shading [Saito and Takahashi 1990] could even reduce the cost for the pixel shader by performing an early z-culling. In our experiments, we found out that if we first render the scene to fill the z-buffer before the final rendering, we get a 20% speed up. This is due to the fact that the driver detects that our shader does not modify the depth of fragments and can perform culling before shading, which is thus evaluated only for visible fragments. At the moment we still lose performance, because shadow maps are shot using pbuffers. Depth recovery using framebuffer objects did not work properly on the tested cards.

Our method is similar to the deep shadow approximation in [Nguyen and Donnelly 2005]. They can have 4 slices without MRT and 16 with MRT. Our approach has 32 slices without and 128 with MRT. On the other hand if one wants to treat partial occlusion via linear interpolation, as they do, it adds some extra cost. Our method approximates the one dimensional transmittance function. The function is evaluated at equidistant samples instead of the non-uniform sampling of the deep shadow maps. Only the first and the last samples are placed at the exact same location.

Our transmittance shadow maps are also closely related to [Bertails et al. 2005]. Here self-shadowing for hair is performed at interactive rates for directional light sources. The authors also create a

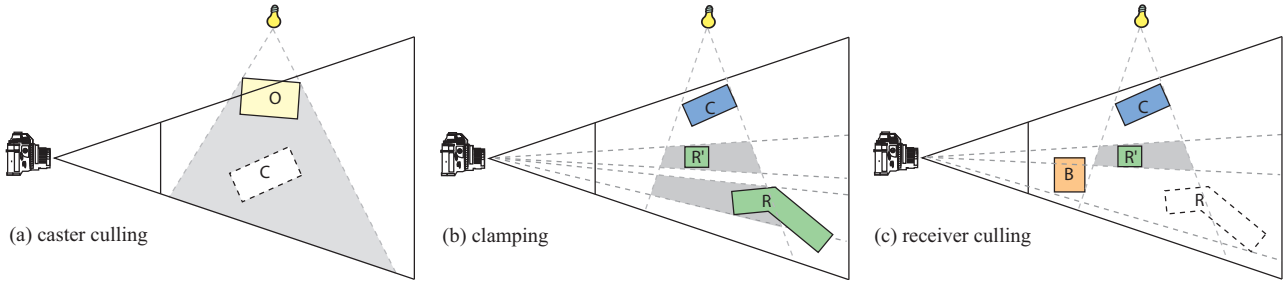


Figure 6: The principle of shadow volume culling and clamping (a) shadow caster C is fully in the shadow of O so its shadow volume can be culled (b) the shadow volume for C need only to extend through regions containing shadow receivers (c) if a shadow receiver R is not visible from viewpoint, the shadow volume for C does not need to be rendered around it.

voxel grid, but theirs is completely uniform. Each hair strand is sampled by points and each point is transferred into the grid via the CPU. One difference is that their voxel grids can contain arbitrary density values, whereas in our method only the presence of matter is detected. On the other hand our grid usually contains more voxels. No point sampling has to be performed, our grid automatically encloses the object tightly and does not have to be adjusted in each frame and we can even treat point light sources.

As future work in this area we would like to see whether our result could be used to approximate soft shadows in a similar manner as [Agrawala et al. 2000] do with layered depth images.

3.2 Shadow Volume Culling and Clamping

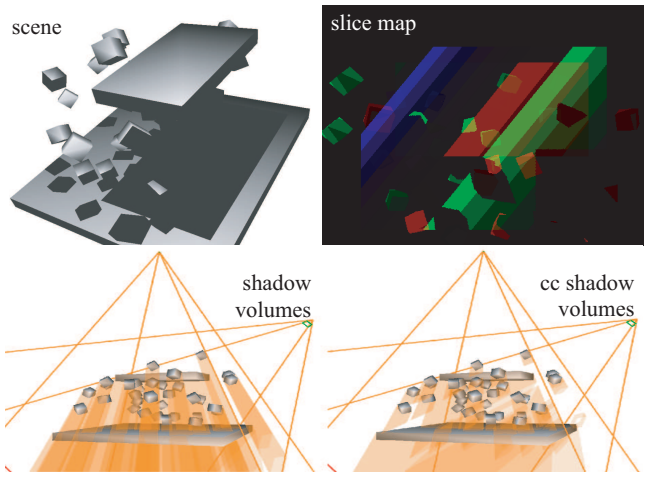


Figure 7: Shadow volume culling and clamping

CC shadow volumes is a technique introduced by [Lloyd et al. 2004] to reduce the fill rate incurred by rendering shadow quads that do not contribute to any shadow in the current view. There are three situations to consider illustrated in Figure 6. Shadow casters that are fully shadowed can be culled as any shadow they would cast will be created by what shadows them. For that [Lloyd et al. 2004] test if a caster is visible from the light source by testing it against a shadow map using occlusion queries. For non-culled casters, the shadow volumes need to be rendered only around receivers that are visible from the light source (including the caster himself). To find those potential receivers, [Lloyd et al. 2004] simply test them from the depth buffer of the observer's view using occlusion queries. To clamp the shadow volumes, the observer's view frustum is cut in n_l layers by planes containing the viewpoint and oriented according to the light direction. The reason for that is that the intersection of front and back facing shadow quads with a layer projects on exactly the same trapezoid in the observer's view so there is no need to project shadow caps on the layer's delimiting planes. For a given layer, [Lloyd et al. 2004] render the potential receivers with the two delimiting planes as clipping planes. Then the projection (from the light source) of each caster on the furthest delimiting plane is rendered with a depth test. If no fragment passes the depth test, the shadow volume can be clamped for consecutive layers. The test is performed using occlusion queries. The receivers are thus rendered n_l times. Note that each of these renderings is not very costly because the clipping planes discard many primitives in the transform stage, but the geometry needs to be send n_l times. The casters are also rendered n_l times leading to a total cost of $n_l(n_r + n_c)$ where n_c and n_r are the number of non culled casters and receivers.

Another method for performing CC shadow volumes is presented in [Décoret 2005] but it cannot do multiple clamping and is not yet fast enough to compete. Slicemaps contain all the information to perform CC shadow volumes in a more efficient way than both previous methods. In this section, we first describe an ideal algorithm that is based on a proposed hardware extension that we believe is really easy to implement on a chipset. We then describe a less efficient implementation that works on current hardware and performs better than [Lloyd et al. 2004]. We also show two ways of even further clamping shadow volumes.

Ideal algorithm As in CC shadow volumes[Lloyd et al. 2004], we first cull casters and receivers by testing them against a shadow map and the observer's depth map respectively. Then we compute a uniform slicemap with the potential receivers. The computation is a bit different than in Section 2 because the slices must follow the layers instead of being perpendicular to the light direction. To find the cell containing a fragment, we no longer use the cellmask texture. Instead, we project the corresponding 3D point into the observer's view and compute its distance to the projection of the light along the projection of the light's direction. This distance can be computed at vertices and correctly interpolated for fragments.

To find where a caster's shadow volume should be clamped, the bitmasks of all texels of the slicemap on which the caster projects need simply to be OR-ed. This could be done by extending the occlusion queries mechanism so that for every fragment rasterized, the content of the framebuffer is OR-ed with the value of a special register (initialized to 0). OR operations are order independent and can be done concurrently so it will not offend the SIMD nature of GPU. The API would then return the value of the register similarly to the number of z-test success returned by occlusion queries. The demand for this kind of reduction register to treat order independent activities exists for quite some time, but in our case the structure would even be simpler, as the value would not even need to be locked while the modification occurs. With such an extension, clamping a caster would only require a single rendering of the caster, no matter the number of slices. The cost for the slicemap is independent of the number of slices, resulting in a total of $n_r + n_c$.

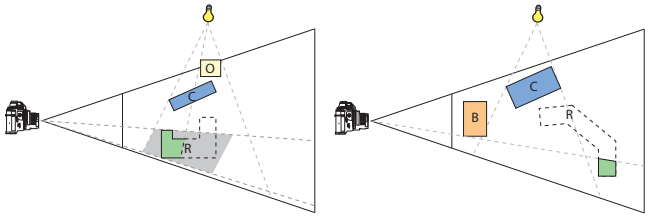


Figure 8: Improved culling and clamping

Practical algorithm Without the proposed hardware extension, we can still use a slicemap for clamping. A first solution would be to perform the OR by hand using a matrix reduction method similar to [Buck and Purcell 2004]. It is a bit tricky to set up and it might be too slow. Instead, we propose to use occlusion queries like [Lloyd et al. 2004]. We run through each slice and use a fragment program to test if the covered texels have the bit for the current slice set to 1. If not, we discard the fragment. We use an occlusion query to find if at least one fragment is not discarded and decide whether to clamp or not. The cost of our approach is n_r for the slicemap generation and $n_l n_c$ for the clamping. So the total cost is $n_r + n_l n_c$ which is less than for CC shadow volumes.

Improved culling and clamping The algorithm we described so far does not perform optimal culling and clamping. Figure 8 shows two cases that are not handled. On the left, the algorithm would cull the shadow volume of C around the whole receiver R. But since a part of R (the one dashed) is actually shadowed by another caster O, the shadow volume can be clamped tighter. To account for this situation, when testing a caster for a slice, we compare the depth of each fragment with the depth in the shadow map. If it is strictly greater, we can safely discard the fragment and ignore its bitmask. In other words, we do pixel-based caster culling as opposed to object-based caster culling. The rightmost example shows a receiver that is visible by the observer but cannot actually reveive shadows. To account for this, we use the litmap approach described in [Décoret 2005]. When generating the slicemap, we discard any fragment that is not visible from the observer. Thus this fragment does not generate any 1 bit in the slicemap. Once again, this amounts to pixel-based instead of object-based receiver culling. Note that these two improvements would work straightforwardly with [Lloyd et al. 2004] although they are not described there.

Results We implemented the algorithm to test its feasability. Figure 7 shows the results obtained on a simple scene. A validation on more complex scenes and an exact measured comparisons could be of interest, but our main intention was to show the applicability and the possible gain of our method (especially when the proposed hardware extension becomes avaiable) with respect to previous approaches.

3.3 Refraction and Frosted Glass

The voxel representation can be used to calculate an approximation of the volume traversed by a ray. Based on this distance refraction can be increased, scattering can be approximated or colors can be shifted towards the color of the object to simulate gas effects.

In [Sousa 2005] a simple approach to obtain reasonable refraction effects has been presented and our work is inspired from this approach. Based on the surface normal where the eye ray hits the object a look-up in an environment texture is perturbed. The idea of taking volume into account has been presented recently in [Wyman 2005]. A normal and a depth map for the closest and the farthest surfaces are calculated and for each vertex of the object a depth along the normal is precalculated. The algorithm perturbs rays based on an interpolation between the precalculated depth and the difference of the depth maps. This information is then used together with the two normal maps to obtain the final ray. One major problem is that using the closest and farthest surfaces to approximate the volume traversed by an eye ray is correct only for convex objects and will give arbitrarily wrong values in other cases.

In this section, we present our solution. It requires no precomputations and uses more reliable information about the actual volume traversed by an eye ray. It simulates three effects : refraction, attenuation and finally scattering related to the traversed volume. Nevertheless assumptions about the input model are necessary. It should be watertight because otherwise no proper volume is defined. Also it has to be possible to determine whether a drawn fragment corresponds to a front or back facing surface.

We make the simplifying assumption, that the volume traversed by the refracted ray is closely related to the volume traversed by the *eye ray* (that is the straight ray coming from the eye). The sum of all faces intersected by the eye ray as in 3.1 would not result in an acceptable volume (e.g. for a cube it would be constant for any point of view as an intersecting ray always hits a convex object two times).

Because our input model is closed, each ray hitting the object will enter via a front facing point on the surface and remain inside the object exactly until a back facing surface is hit [Crow 1977]. The ray then remains outside the object until a new front facing surface is encountered and the process repeats itself. The complete traversed volume is then the sum of all intervals from a front facing to a back facing surface along the ray.

To obtain the information about the face's orientation, two local slicemaps are created. The first considering only the front, the second only the back facing polygons. This can be done in a single pass using MRT and a fragment shader that outputs the fragment in the according texture depending on its orientation. To achieve coherence between the two representations we use the same local normalization for both textures.

Although the precision could still be increased via the other two channels, we found in practice that 32 slices (therefore one texture for each orientation) with local precision suffices to obtain convincing results.

With the two slicemaps, we want to find for each ray the number of traversed *inner voxels* (those lying inside of the volume defined by the model). This number, scaled according to the local precision obtained from the depth textures, results in the volume approximation. We will call a *front/back voxel* a filled voxel in the front/back slicemap.

A naive approach would sum up all voxels between front and back voxels. There are several problems concerning this simple approach. First in some situations it might completely fail, due to the discrete nature of the representation and second it does not map well onto graphics hardware. Let's first look at the situations in which the algorithm fails.

The main problem is that front and back facing surfaces can fall in the same voxel which we then qualify as *ambiguous*. Conversely an *unambiguous* voxel contains only one type of orientation. Ambigous voxels raise two problems shown on Figure 9. For such voxels, it is impossible to retrieve the traversed volume. We address this by simply considering the whole voxel as an inner one, which is a reasonable approximation given its size. The second problem is that within an ambiguous voxel, the object's surface is crossed an unknown number of times. Thus, the following empty

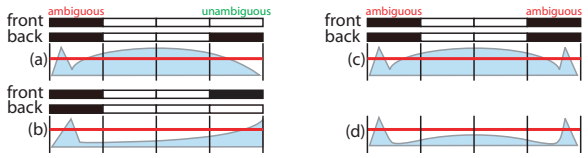


Figure 9: (left) The status of empty voxels in between an ambiguous and unambiguous voxel can be resolved as inside if the unambiguous voxel is front (a) and outside if it is back (b). For empty voxels in between two ambiguous voxels, the bitmasks cannot disambiguate between inside (c) and outside (d).

voxels could be inner voxels or not. This remains unresolved until a non empty voxel is encountered. If it is unambiguous, the previous voxels state can be determined. If it is ambiguous, nothing can be said. In practice we decide not to include the voxels between. This is potentially a wrong classification but two successive ambiguous voxels are actually rare. Moreover, if there are not two front faces in an ambiguous voxel, the choice is correct.

Creating an efficient fragment program to perform these operations on the GPU is not trivial. Recovering all bits and applying the naive algorithm would be too slow. The fastest possibility to solve the voxel integral would be to use a texture lookup giving the number of inner voxels for a given bitmask. Since there are two bitmasks of 32 bits, the texture should be a 2D table of $2^{32} \times 2^{32}$ entries, that is approximately 16 GB.

The observation is that the naive algorithm is incremental; calculating the number of inner voxels at position $i+1$ only requires the knowledge of the number of inner voxels traversed up to position i and the *state* (whether the current position is inside or outside) after i. Thus it is possible to treat each channel successively. An evaluation texture for a single channel now requires only 256×256 entries. However we still have a dependency on the state (theoretically leading to dependent texture lookups) and we need to recover the new state at the end of the channel. Since the number of inner voxels is at most 8, only 4 bits are needed to encode it and we can use the 4 remaining bits to encode the new state (we simply add 128 if it remains inside). Volume and new state still depend on the old state. To solve this we store the two possible solutions in two channels of the evaluation texture. The two remaining channels will be used to treat ambiguity. We store whether the first non empty voxel is a back voxel or not and if the last non empty voxel is ambiguous, one plus the number of succeeding empty voxels.

This data is used to compute the number of inner voxels in the following manner. We first fetch the four RGBA values from the evaluation texture for the 4 pairs of front/back bitmasks. (No dependent texture lookup is necessary.)

For each pair i successively, if the previous state is inside, add value of R_i to the number of inside voxels, otherwise add value of G_i. In both cases, update the state accordingly (R_i or $G_i > 128$). Then if $A_i \neq 0$, successive channels must be checked to resolve the ambiguity. This is done by looking at B_{i+1}, as it contains the type information on the first non empty voxel of the channel (in the special case where all voxels in channel $i+1$ are empty we proceed with $i+2$ etc.). Once the ambiguity is resolved A_i can be used to add the voxels behind the ambiguous position. Once the last channel is treated, the final volume is computed as the sum of all inner voxels normalized by the local depth interval.

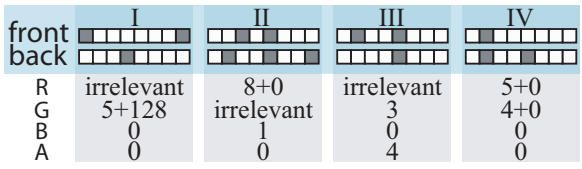


Figure 10: Examples of evaluation to RGBA values for pairs of bitmasks."Irrelevant" represents situations that cannot occur. (e.g. encounter a front face, inside of the object)

Figure 11: Various materials with different refraction/absorption parameters. Images are obtained at $> 200 fps$ in a resolution of 512×512 on a Geforce 6800 Ultra (cf. color plate BII)

To make the refraction dependent on the volume, a simple scaling is applied to the offset perturbation of the environment map lookup. To simulate attenuation, an opacity is calculated based on a power function. For the scattering effect, we used a hierarchical environment map built either as a mipmap pyramid [Heckbert 1986] or according to [Hensley et al. 2005] for better results and less sampling artifacts. Depending on the traversed volume a higher or lower level of the pyramid is accessed for the perturbed refraction lookup. This leads to blurrier information the more the ray stays inside the object. Figure 11 shows the effects we obtain and figure 12 summarizes the approach.

The frame rate is far above 200 fps. Our refraction model being quite simple, some artifacts are observable. For example, parts of the object behind others will shine through with exact borders. Nevertheless we found this very acceptable especially in comparison with a simple volume estimation only based on depth maps, which yields an unrealistic appearance in all concave regions. In our implementation, all parameters for refraction indices, color, absorption can be dynamically changed. If these parameters are intended to be constant, the shaders could be simplified and the frame rate would even increase.

The presented model is a first step in the direction of exploiting the voxel representation for volumetric effects. It could be interesting to combine our approach with translucent shadow maps [Dachsbacher and Stamminger 2003], or to write the volume values in a first step in a texture, where they could be further processed (e.g. filtered), which would then allow the object to have an influence on itself.

4 Conclusion and future work

We presented a new method to quickly calculate a simple voxel representation using graphics hardware. The method is very fast -milliseconds for complex models- and the resolution of the voxel grid is high in 2 dimensions and poorer, though reasonable in the third. We introduced local precision, a feature which is perfectly suited for the use on graphics cards, along with MRT to increase this resolution. Our approach benefits from the structure of the graphic cards and can integrate with shaders to produce various effects. Several applications have been presented, which, in our eyes, are interesting contributions to the community on their own. Our *transmittance shadow maps* allow emulation of simple deep

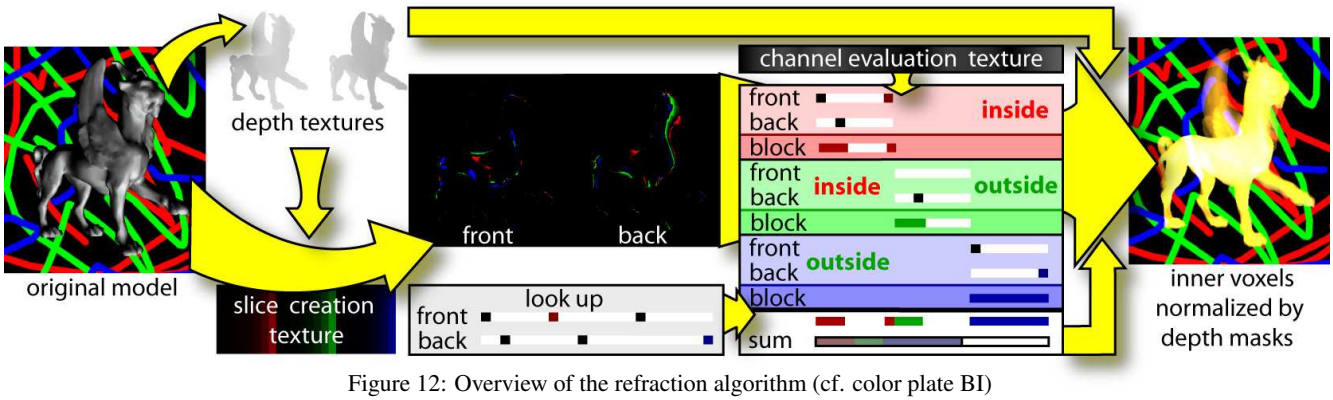


Figure 12: Overview of the refraction algorithm (cf. color plate BI)

shadow maps even on older hardware with an acceptable precision and frame rates. Newer hardware supporting MRT improve the quality of our approximation at almost no cost. Several effects can be achieved, based on the information of the object's volume. Our approach combines the advantage of exact shape information in the form of depth maps and approximated volume in the form of voxels and results in a good overall estimation. We have shown that this information could e.g. be used to create frosted glass like effects. Other applications are possible and we are currently investigating e.g. fast particle collision detection. Error estimations and conservativity also remain future research.

We believe that our algorithm will become even more valuable on future graphics cards. In particular when 32 bit precision logical operations become possible, the number of slices could be increased to 512 (32 bit times 4 channels times 4 render targets) instead of currently 128 (8 bit) in a single pass and at almost no supplementary cost.

Acknowledgements We thank M. Kaplan, E. Turquin, M. Eisemann, E. Chan for their remarks and the reviewers for their insightful comments. The work was founded by the region Rhône-Alpes (Dereve) and the École Normale Supérieure Paris.

References

AGRAWALA, M., RAMAMOORTHI, R., HEIRICH, A., AND MOLL, L. 2000. Efficient image-based methods for rendering soft shadows. In *SIGGRAPH 2000: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*.

AILA, T., AND AKENINE-MÖLLER, T. 2005. Conservative and tiled rasterization. *Journal of Graphics Tools 10(3)*.

BERTAILS, F., MÉNIER, C., AND CANI, M.-P. 2005. A practical self-shadowing algorithm for interactive hair animation. In *Graphics Interface*. Best student paper award.

BRESENHAM, J. E. 1965. Algorithm for computer control of a digital plotter. *IBM Systems Journal 4*, 1.

BUCK, I., AND PURCELL, T. 2004. *GPU Gems*. Addison-Wesley, ch. Ch. 37: A toolkit for Computations on GPUs.

CROW. 1977. Shadow algorithms for computer graphics. In *Proceedings of SIGGRAPH '77*.

DACHSBACHER, C., AND STAMMINGER, M. 2003. Translucent shadow maps. In *Proceedings of Eurographics Workshop on Rendering '03*.

DÉCORET, X. 2005. N-buffers for efficient depth map query. *Computer Graphics Forum 24*, 3.

EVERITT, C., 2001. Interactive order-independent transparency. http://developer.nvidia.com/object/Interactive_Order_Transparency.html .

HAUMONT, D., AND WARZEE, N. 2002. Complete polygonal scene voxelization. *Journal of Graphics Tools 7*, 3.

HECKBERT, P. S. 1986. Survey of texture mapping. In *IEEE Computer Graphics and Applications*.

HENSLEY, J., SCHEUERMANN, T., COOMBE, G., SINGH, M., AND LASTRA, A. 2005. Fast summed-area table generation and its applications. In *Proceedings of Eurographics '05*.

KIM, T.-Y., AND NEUMANN, U. 2001. Opacity shadow maps. *Eurographics Rendering Workshop*.

LEFEBVRE, S., HORNUS, S., AND NEYRET, F. 2005. Texture sprites: Texture elements splatted on surfaces. In *Symposium on Interactive 3D Graphics (I3D)*, ACM Press, ACM SIGGRAPH.

LLOYD, B., WENDT, J., GOVINDARAJU, N. K., AND MANOCHA, D. 2004. Cc shadow volumes. In *Proceedings of the 2nd EG Symposium on Rendering*, Eurographics Association, Springer Computer Science, Eurographics.

LOKOVIC, T., AND VEACH, E. 2000. Deep shadow maps. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.

NGUYEN, H., AND DONNELLY, W. 2005. *GPU Gems 2*. Addison-Wesley, ch. Ch. 23: Hair Animation and Rendering in the Nalu Demo.

SAITO, T., AND TAKAHASHI, T. 1990. Comprehensible rendering of 3-d shapes. In *In SIGGRAPH (Proceedings of the 17th annual conference on Computer graphics and interactive techniques)*.

SHADE, J., GORTLER, S., WEI HE, L., AND SZELISKI, R. 1998. Layered depth images. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, ACM Press, New York, NY, USA.

SOUSA, T. 2005. *GPU Gems 2*. Addison-Wesley, ch. Ch. 19: Generic Refraction Simulation.

WILLIAMS, L. 1978. Casting curved shadows on curved surfaces. In *SIGGRAPH '78: Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, ACM Press, New York, NY, USA.

WYMAN, C. 2005. An approximate image-space approach for interactive refraction. In *SIGGRAPH 2005: Proceedings of the 32th International Conference on Computer Graphics and Interactive Techniques*, ACM Press, New York, NY, USA.